**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

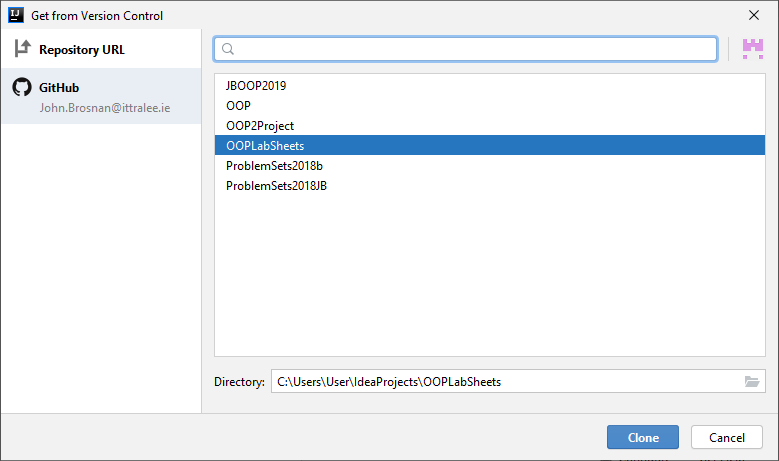
**Practical 8 – Inheritance**

At this stage you have covered many of the basic concepts in OOP. As you saw last time, **composition** and **aggregation** are central to the whole notion of **software reuse**. This lab sheet covers **inheritance**, which is the other main OO feature than enables us to take advantage of existing software components, sparing us great time and effort.

**Getting into IntelliJ**

Launch IntelliJ. As you were introduced to **VCS** already, and since you should really have the latest version of your **OOPLabSheets** project “pushed” to GitHub, Click “**Get from Version Control**” and see if you can now clone your **OOPLabSheets** project locally (if you haven’t your latest version pushed to GitHub, just copy it from your your X: drive to some location on C: or work directly with it from the memory stick).

IntelliJ will remember previously “pushed” repositories, so you can just pick off the one you want. You can also decide where you want the repository to be located locally, I am choosing the same location as the original,



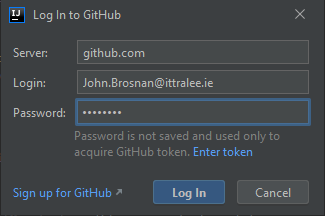
Recall that you **may need to rename** the local repository to something else if you get an error message at this point (I suggest a date e.g. OOPLabSheets15-10-20).

**Pushing an IntelliJ Project to GitHub**

**At the end of the lab session** you should really “**push**” your version-controlled OOPLabSheets project to GitHub, so that it then becomes a remote repository and a back-up of your work. I suggest you back it up to X: or memory stick also.

To do this, within IntelliJ just select **VCS**, then **Git**, then **Push** or, even better, just press the Git “Push” green arrow at the top of the IntelliJ window.

At this point, a dialog may pop up requesting your GitHub account details (IntelliJ might remember these also though). Once supplied, you can press the **Log In** button



If everything goes to plan, you will get a “Pushed 1 commit to origin master” message at the bottom-right of the IntelliJ window:

Now, for proof that the project is actually on GitHub, you can just **view your GitHub repositories list** and you should see the **OOPLabSheets** project listed. You can click into this then to make sure the latest files are definitely there.

**Setting up your Folder Structure**

My own preferred approach is to create a new folder for each lab sheet. In IntelliJ this can be done by adding a new **package** to the project. The package will be given an appropriate name, I will call it **labsheet8** here. Recall that **a Java package is simply a way to store related classes together** and essentially a **package is just a folder**. We will talk about packages further in this module but, for now, we will just create a package called **labsheet8** for this IntelliJ project and our intention will be to store all the related classes that we create and use for this lab sheet together within that package (folder). Right-click on the name of the project i.e. **OOPLabSheets** and select **New**🡪**Package.**

You will now be given the opportunity to enter the name of the package, so you can enter **labsheet8**. As soon as you click **OK**, an icon for the newly created package appears in the left-side window, listed as part of the project’s contents. The package is currently empty, but you will be adding some sub-packages to it as the lab goes on.

**Inheritance in Java**

**Inheritance** is a form of **software reuse** that enables us to create new classes which are based on existing ones. The new class, called the **derived class** or **subclass**, will subsume the members of the existing class and can directly access all of the **non-private attributes and methods** (bar constructors) which are defined in the existing class, called the **base class** or **superclass**. The subclass then goes on to **add some of its own attributes and methods** to distinguish it from its superclass. It promotes **extensibility** by allowing us to “extend” the functionality of existing classes, **reduces code duplication** and allows **less complex classes** to be created.

The opportunity to use the concept of inheritance exists when you can identify a situation where you have 2 entities where an **“is-a” relationship** exists e.g. a Student **is a** Person, a Car **is a** Vehicle, a FacultyMember **is an** Employee etc. So Student, Car and FacultyMember would be the subclasses here.

The diagram below illustrates an **inheritance hierarchy** for a number of entities. You can see that such a hierarchy can have several layers if required.

bjlo_ch09_vehicles.pdf

General

More

Specialised

Specialised

So you can see that **Vehicle** is at the top of the hierarchy, as it is the **most general** specification. A Vehicle object could be any vehicle you care to think of, including bicycles, horse & carriage, rickshaws etc. Beneath this we have several classes of Vehicle such as Motorcycle, Car and Truck. These are certainly all vehicles, so an **is-a** **relationship** definitely exists between them and the superclass Vehicle. However, these particular classes of vehicle all **have their own specific features** that make them **more specialised** than a Vehicle and so can be distinguished from it (and each other). Then, at the bottom level, you can see a number of Car subclasses such as Sedan and SUV. Certainly each of these **is-a** Car, but each of them has their own specific properties that distinguish them from and make them **more specialised** than their more general superclass.

The following sample program **demonstrates inheritance** where there are two instantiable classes representing a Student and a Person.

**Instantiable class Person**

package labsheet8.sampleprogram1;  
  
*//Person.java  
/\*An instantiable Person class that acts as a superclass for the Student class\*/*import java.util.Calendar;  
import java.util.GregorianCalendar;  
  
public class Person {  
 private String name;  
 private String address;  
 private GregorianCalendar dateOfBirth;  
  
 public Person(){  
 this("No name specified","No address specified",null);  
}

public Person(String name, String address, GregorianCalendar dateOfBirth){  
 setName(name);  
 setAddress(address);  
 setDateOfBirth(dateOfBirth);  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getAddress() {  
 return address;  
 }  
  
 public void setAddress(String address) {  
 this.address = address;  
 }  
  
 public GregorianCalendar getDateOfBirth() {  
 return dateOfBirth;  
 }  
  
 public void setDateOfBirth(GregorianCalendar dateOfBirth) {  
 this.dateOfBirth = dateOfBirth;  
 }  
  
 public String toString(){  
 String str = "Name: " + getName() + " Address: " + getAddress() + " Date of Birth: ";  
  
 if(dateOfBirth==null)  
 str+="No date of birth specified";  
 else  
 str+=getDateOfBirth().get(Calendar.*DATE*) + "-" + getDateOfBirth().get(Calendar.*MONTH*) +  
 "-" + getDateOfBirth().get(Calendar.*YEAR*) ;  
  
 return str;  
 }  
}

**Analysis of the Person Class:**

● The Person class is a typical looking instantiable class containing a no-argument and multi-argument constructor, a set of mutators and accessors for each of its three attributes and a toString().

● One complication here is the presence of the dateOfBirth attribute which is of type **GregorianCalendar**. This is a **Java API class** which comes from the **java.util** packageand is **used to create and manipulate dates** to great effect. As it is a non-primitive type, then we can say here that a Person object “**has-a**” GregorianCalendar object **nested** within it and so it is a very good example of **composition**, since the date of birth does not “live” independently of the Person object it is associated with.

● The presence of the dateOfBirth attribute also complicates things a little in the toString() method. The accessor is used here to get us the reference to the GregorianCalendar object, but then this must be manipulated to “pull out” the day, month and year parts individually. This is done by calling the **get**() method on the reference and passing in the various **constants** you can see as arguments e.g. the constant **Calendar.DATE** allows us to pull put the “day” part of the date of birth value.

**Instantiable class Student**

package labsheet8.sampleprogram1;  
  
*//Student.java  
/\*An instantiable Student class that inherits from a Person superclass\*/*import java.util.GregorianCalendar;  
  
public class Student extends Person {  
  
 private int id;  
 private String dept;  
  
 public Student(){  
 setId(0);  
 setDept("No department specified");  
 }  
  
 public Student(String name, String address, GregorianCalendar dateOfBirth, int id, String dept) {  
 super(name,address,dateOfBirth);  
 setId(id);  
 setDept(dept);  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getDept() {  
 return dept;  
 }  
  
 public void setDept(String dept) {  
 this.dept = dept;  
 }  
  
 public String toString(){  
 return super.toString() + " ID: " + getId() + " Department: " + getDept() + "\n";  
 }  
}

**Analysis of the Student Class:**

● The first thing to talk about here is the class definition header which looks as follows:

public class Student extends Person {

The word **extends** is a Java **keyword** that you won’t have seen in first year. Its presence always means that an **inheritance relationship** is being defined. The general form is:

**Access-modifier class *SubClassName* extends *SuperClassName* {**

So we can say here that the Student class is being defined as a subclass of the Person class (which will be its superclass)

● From a modelling point of view, of course this relationship makes logical sense, as a Student definitely **is-a** Person. It means that all the non-private attributes and methods (bar constructors) that have been defined in the Person class are now available directly within the Student class.

In this case, all the attributes defined within the Person class are private, meaning that none of those are directly available within the Student class. However, all of the mutators and accessors, along with the toString() are available, since they are public.

● The no-argument constructor is defined as follows:

public Student(){  
 setId(0);  
 setDept("No department specified");  
 }

It is invisible here, and certainly not at all obvious, but the very first thing the Student constructor does here is to **make a call to the superclass’ no-argument constructor**. This is because the attributes defined within the Person class need to be initialised for any Student object that will be created.

It would be **illegal** to do the following though:

public Student(){

Person(); //would generate a syntax error!  
 setId(0);  
 setDept("No department specified");  
 }

Instead Java has another keyword called **super**, that can be used to explicitly call a superclass no-argument constructor, but **only from a subclass constructor**. It could be done as follows here:

public Student(){

super(); //this is allowed but, if omitted, it will happen anyway  
 setId(0);  
 setDept("No department specified");  
 }

So, once the call to super() occurs, the Person no-argument constructor will be called and the three attributes defined within it will be given initial values.

● The multi-argument Student constructor looks as follows:

public Student(String name, String address, GregorianCalendar dateOfBirth, int id, String dept) {  
 super(name,address,dateOfBirth);  
 setId(id);  
 setDept(dept);  
 }

Again, the first thing that happens when this constructor is called is that a **call to the Person multi-argument constructor** will occur. The Student constructor here takes five arguments representing the five properties that define it, three of which are defined within Person. Once it receives the five values, three of them are **passed “up”** to the superclass in order for it to set the initial values of the attributes defined within it.

So here the name, address and dateOfBirth arguments are all passed on to the Person constructor and it has the responsibility to set those attributes up with values.

Note that the call to **super**() is explicit this time and it **will not be done automatically**, unlike in the case for the no-argument constructor.

Note also that the **call to super() must always be the very first statement in a subclass constructor**, otherwise a **syntax error** will occur.

● The remainder of the Student class looks normal until we come to the toString() method, defined as follows:

public String toString(){  
 return super.toString() + " ID: " + getId() + " Department: " + getDept() + "\n";  
}

You can see here that the **super** keyword is evident again. However this time, rather than being used to call a superclass constructor, it is being **used as an object reference** on which the toString() method of the Person superclass is called. So the Student toString() makes a call to the Person toString() in order to complete its task. Software reuse at its finest! We could go to the trouble of calling all the accessors defined within the Person class one by one from the Student toString() but that would be madness, when we can simply call the Person toString() to get the necessary information. But, to distinguish between the toString() of the local class (Student) and that of the superclass, the super keyword is necessary.

**Driver Class TestPerson**

package labsheet8.sampleprogram1;  
  
*//TestPerson.java  
/\*This driver program tests out the classes Person and Student fully\*/*import javax.swing.\*;  
import java.util.GregorianCalendar;  
  
public class TestPerson  
{   
 public static void main(String[] args)   
 {   
 String text;  
   
 text = "Calling the Person() constructor .....";   
 Person person1 = new Person();  
 text += "\nValue of first Person object is: " + person1;  
  
 text += "\n\nCalling the Person(String,String,GregorianCalendar) constructor after getting user-supplied values .....";  
   
 String name = JOptionPane.*showInputDialog*("Please enter the name of the second person");  
 String address = JOptionPane.*showInputDialog*("Please enter the address of the second person");  
 String dateOfBirth = JOptionPane.*showInputDialog*("Please enter the date of birth of the second person in the form dd-mm-yyyy");  
  
 int day = Integer.*parseInt*(dateOfBirth.substring(0,2));  
 int month = Integer.*parseInt*(dateOfBirth.substring(3,5));  
 int year = Integer.*parseInt*(dateOfBirth.substring(6,10));  
  
 GregorianCalendar birthDate = new GregorianCalendar(year,month,day);  
  
 Person person2 = new Person(name, address,birthDate);  
 text += "\nValue of second person object is: " + person2;  
  
  
 text += "Calling the Student() constructor .....";  
 Student student1 = new Student();  
  
 text += "\nValue of first Student object is: " + student1;  
  
 text += "\n\nCalling the Student(String,String,GregorianCalendar,int,String) constructor using hard-coded values .....";  
 Student student2 = new Student("Michael O' Connor","Ballybunion, Co. Kerry",  
 new GregorianCalendar(1996,4,12),425362,"Computing");  
  
 text += "\nValue of second Student object is: " + student2;  
  
 JOptionPane.*showMessageDialog*(null,"\*\*\*\*\*Person and Student Class Tester\*\*\*\*\*\n\n" + text,"",  
 JOptionPane.*INFORMATION\_MESSAGE*);  
   
 System.*exit*(0);  
 }   
}

**Analysis of the TestPerson Class:**

● The first part of this driver tests out the functionality of the Person class, which is acting as a superclass for the Student class. The no-argument and multi-argument Person constructors are called in turn to create two Person objects. Their state is displayed later on, with the state of the Student objects.

● The multi-argument constructor takes a reference to a GregorianCalendar object as an argument. In order to create the GregorianCalendar object here, we take in the date of birth on an input dialog. Of course, this value will be returned as a piece of text and is stored in the local String variable dateOfBirth. The **substring**() method, that we used regularly last year, is then used to “pull out” the various parts of the date of birth: the day, the month and the year. Once all of these have been converted to integers, we are then in a position to call the GregorianCalendar constructor as follows:

GregorianCalendar birthDate = new GregorianCalendar(year,month,day);

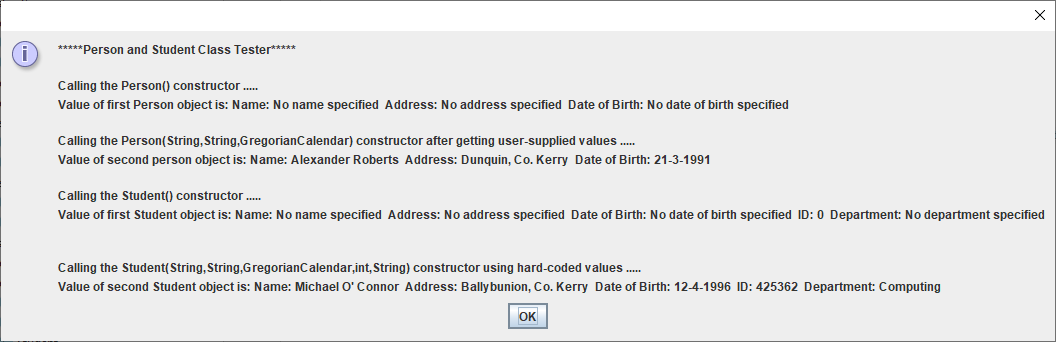
Note that when the constructor is called, the **year** **must be the first argument** supplied, rather than the day, which is often a source of logical errors.

● The next part of the program involves creating two Student objects. Notice the call to the multi-argument constructor involves 5 arguments, to account for the extra three provided by its superclass Person. You’ll also notice that here I just put the creation of the GregorianCalendar object directly within the constructor call as follows:

Student student2 = new Student("Michael O' Connor","Ballybunion, Co. Kerry",  
 new GregorianCalendar(1996,4,12), 425362, "Computing");

Rather than create it separately and have another reference variable linked to it (this is better from an encapsulation viewpoint, as the only reference linked to the object is coming from the Person class).

● Finally, the state of all the Person and Student objects are displayed on a message dialog, with the following output:



**View of Participating Classes (VOPC) Diagram**

You have seen UML VOPC diagrams a number of times now. These diagrams show, in graphical form, the **class diagrams** for an application and the **relationships** between the classes defined by those diagrams. You have seen examples of **dependency** relationships, **composition** relationships, and **aggregation** relationships. This time, you will see an **inheritance** **relationship**, based on the example above.

* The **dashed lines** with the arrow at the end, connecting the driver class TestPerson to the other two classes indicate a **dependency relationship** between the driver and those two classes, since it creates objects from both classes, and a change it the definition of either class could necessitate a change in the driver class which is using it. The arrow points to the classes that the other class depends on.
* The solid line with an **filled arrowhead** indicates an **inheritance relationship** between the Student and the Person classes. The class touched by the filled arrowhead is the **superclass** in the relationship, while the other class will be the **subclass**.

A screenshot of a cell phone

Description automatically generated

Copy the **sampleprogram1** package to your **labsheet8** package now. This gives you your own copy of the three classes. To begin with, **compile and run** the application to see its output.

**protected Attributes and Methods**

We looked at **access modifers** in a previous lab sheet in the form of **public**, **private** and **package** access. These are certainly the most commonly used ones in practice, but there is another form called **protected** **access**, which exists **especially for inheritance** relationships between classes.

In the Person-Student example we looked at, you can see that the attributes defined within Person are all private, with all the methods defined public, giving us a **truly encapsulated** entity. It means that the **attributes of the Person class cannot be accessed directly outside** of that class, even by the Student subclass, which is closely related to it.

However, we were still able to get at all the attributes defined within Person **indirectly**, or even set them, via the public accessors and mutators defined within Person, which are also available, via inheritance, to Student of course.

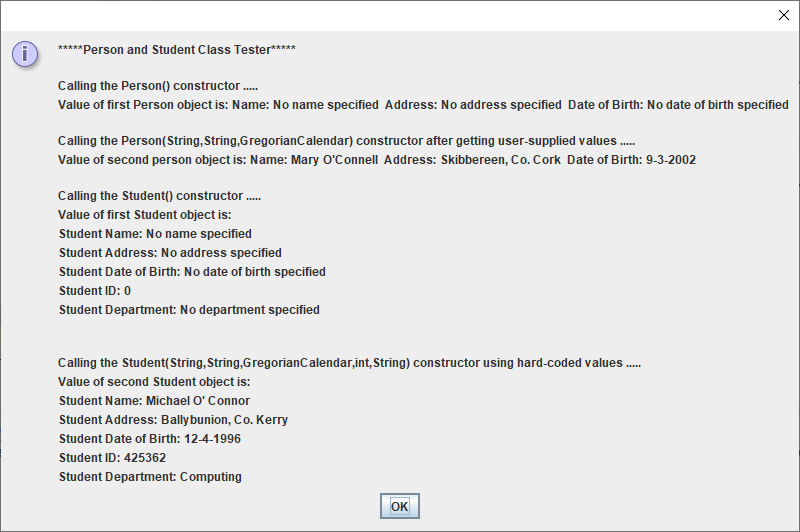
If there was a desire to be able to access the attributes of the Person class **directly** from Student, then we could use the protected keyword when defining the Person attributes as follows:

public class Person {  
 protected String name;  
 protected String address;  
 protected GregorianCalendar dateOfBirth;

Everything else in the Person class could remain exactly the same as before. The difference now is that, because the attributes are protected, they **can be accessed from the subclass directly** as follows, rather than using the superclass accessors:

public String toString(){  
 String str = "Student Name: " + name + "\nStudent Address: " + address + "\nStudent Date of Birth: ";  
  
 if(dateOfBirth==null)  
 str+="No date of birth specified";  
 else  
 str+=dateOfBirth.get(Calendar.*DATE*) + "-" + dateOfBirth.get(Calendar.*MONTH*) +  
 "-" + dateOfBirth.get(Calendar.*YEAR*) ;  
  
 str+= "\nStudent ID: " + getId() + "\nStudent Department: " + getDept() + "\n";  
  
 return str;  
}

So I have replaced the simple call to super.toString() with a modified version of the code contained within that method, which now displays the state of a Student object in a different manner, with the attribute values one per line, rather than them all on the one line as they were before, as you can see below:



Note that this could have been achieved though, by just using the public accessors defined in Person instead, so we could have done the same thing even leaving the attributes in Person private.

So the use of the protected keyword is **very limited** in practice when a class/superclass is defined in the “traditional” OO style having a set of public accessors and mutators. However, if you (or someone else ) design a superclass that doesn’t have any accessors, and you want to tailor your subclass toString() method, for example, then protected access is crucial, as there is no other way to access the superclass data, from your subclass.

Note also that protected access also means that the attributes/methods marked protected are **available to other classes in the same package**.

**Methods** **can also be defined protected**, for example, the accessor and mutator defined in Person could be defined as follows:

protected String getName() {  
 return name;  
}  
  
protected void setName(String name) {  
 this.name = name;  
}

Again, these methods would still be available to the Student subclass and any other subclass of Person, even those subclasses defined in different packages to it.

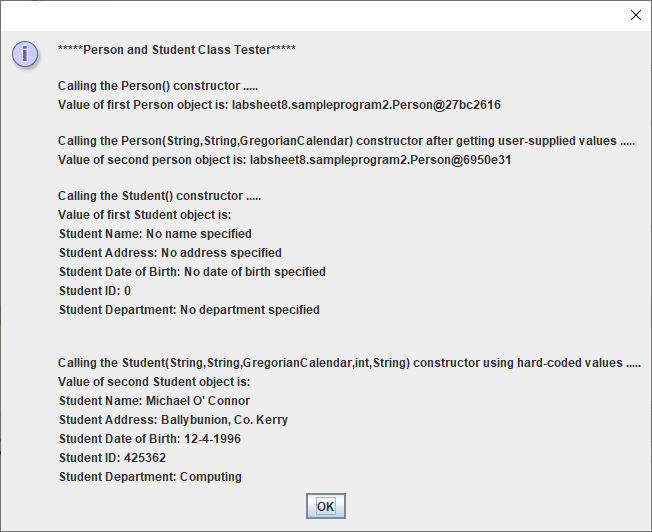
Choosing to make an attribute/method protected is a **design-decision**. Doing so **breaks “true encapsulation”**. If you are delivering an application in binary form and you know that the end-user won’t be “messing” with your code, then it is safe enough to use the protected modifier. The issue would be if a customer was working off your API and tried to subclass from one of your classes that had protected members – if you make these private at a later stage, then their code would require changing.

The **sampleprogram2** package contains an amended version of the earlier example that uses protected attributes and some protected methods. Copy this to your **labsheet8** package now and then **compile and run** the application to see its output.

**Overriding**

You might recall that we discussed this OO concept in an earlier lab sheet. Overriding basically means **redefining a method that is inherited** from a superclass with your own version of the method, to ensure appropriate implementation. It came up earlier in relation to the **toString**() method that is provided automatically by the **Object** class, defined within the Java API, for every Java object.

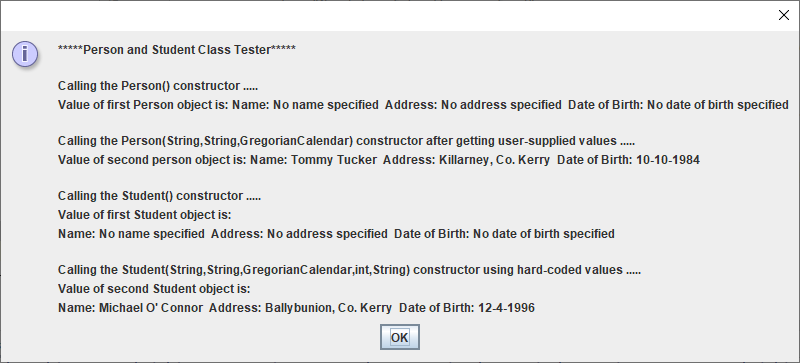
This “default” toString() gives us some strange looking information when we display the state of an object e.g. if I were to comment out the toString() method defined in the Person class of **sampleprogram2**, then the toString() from the Object class would be used automatically and I would get the following output:



Instead of getting meaningful values for the state of the Person objects, I now just get the **package name/class name** and the object’s **hashcode**, which is of limited use to us.

This is the very reason why we need to write our own toString() method for instantiable classes, to **override** (replace) the one that would be used from the Object class.

A similar thing is happening with the Student-Person classes. Here, the Person class defines a toString() method and that same toString(), because it is inherited by Student, would automatically be used by the Student class if it did not override it with its own version. This would mean that, when displaying the state of a Student, we would only be able to see the values of the three attributes defined within Person, as you can see in the screenshot below, where I commented out the Student toString():



**final Methods and Classes**

We used the **final** keyword many times in first year, in relation to **creating constants**. The principle of using the keyword with methods and classes is quite similar. If a **method** is defined final, then it means that it **cannot be overridden** within a subclass. In Java, methods which are private or static are **implicitly final** also. Does this make sense to you? If in doubt about why, feel free to seek explanation.

In a similar manner, a **class that is marked final cannot act as a superclass** at all.

As a quick exercise, now modify the Person class so that it is marked final as follows:

public final class Person {

and **recompile** the application to impress on yourself that there is an issue now.

As another quick exercise, remove the final keyword again from the Person class and make the toString() method within Person final instead as follows:

public final String toString(){  
 String str = "Name: " + getName() + " Address: " + getAddress() + " Date of Birth: ";  
  
 if(dateOfBirth==null)  
 str+="No date of birth specified";  
 else  
 str+=getDateOfBirth().get(Calendar.*DATE*) + "-" + getDateOfBirth().get(Calendar.*MONTH*) +  
 "-" + getDateOfBirth().get(Calendar.*YEAR*) ;  
  
 return str;  
}

**Recompile** the application again to demonstrate to yourself that there is now an issue.

Marking a method or class as final is, like most things in OOP, a **design decision**. Indeed, many Java API classes are final, such as the String class, the Math class, the Integer class etc. This means we **could never inherit** from them.

**Exercise 1**

Now create a **new package** called **exercise1**. You will be writing an application based on the UML VOPC diagram on the next page. You should code your classes so that they **maximise reusability** and also you should write a **minimalistic driver program** that tests out the functionality of your classes.

Note here that, as indicated in the VOPC diagram, there is no dependency relationship between the driver class and the Vehicle class. This just means that you will not be creating any Vehicle objects within the driver here. It is sufficient for you to create Car and Bicycle objects, since these will reuse the Vehicle class anyway and prove that it is functioning as required.

A screenshot of a cell phone

Description automatically generated

Testing the Car no-arg constructor:

Price: 0.0

Length: 0.0

Height: 0.0

Weight: 0.0

Manufacturer: No manufacturer specified

Model: No model specified

Max passengers: 0

Registration Number: No registration number specified

Testing the Car multi-arg constructor:

Price: 100000.0

Length: 4.5

Height: 1.25

Weight: 2000.0

Manufacturer: Ferrari

Model: F2

Max passengers: 2

Registration Number: 05KY1

Testing the Bicycle no-arg constructor:

Price: 0.0

Length: 0.0

Height: 0.0

Weight: 0.0

Manufacturer: No manufacturer specified

Model: No model specified

Has bell: No

Gear Count: 0

Testing the Bicycle multi-arg constructor:

Price: 500.0

Length: 1.5

Height: 1.0

Weight: 50.0

Manufacturer: Raleigh

Model: Mountainbike

Has bell: Yes

Gear Count: 4

**Designing for Inheritance**

It isn’t always easy to know when you can apply inheritance, and it **can easily be misused** or overused. What you are looking for is to identify a set of entities that all share a number of **common properties and operations**, but are sufficiently different from each other to merit creating an **inheritance hierarchy**. The **common features** are then placed in the **superclass**, while the parts that make the entities different from each other are placed in the subclasses. This ultimately allows **less complex classes** to be created, while **promoting software reuse** and **removing duplication**.

**Single versus Multiple Inheritance**

Java only has support for **single inheritance** which means that a (sub)class can only directly inherit from a single superclass e.g. even though a Lecturer **is-a** Employee and also is-a Person, a Lecturer cannot inherit from both Employee and Person simultaneously. In this case though, you could get around the issue by noting that a Lecturer **is-a** Employee and an Employee **is-a** Person so two instances of single-inheritance allow the Lecturer to inherit from the Person class.

Employee

Person

Lecturer

In other languages, such as C++, there is support for **multiple inheritance** where a class can directly inherit from any number of superclasses e.g. a StudentTeacher inherits from a Student but also from a Teacher class.

The main reason Java omits this OO feature is to **keep things simple** and straightforward and to avoid well-known issues such as the so-called “Diamond Problem”.

Java does **provide the advantages of multiple inheritance through the notion of interfaces** however, which will be looked at later.

Note that a Java **class can be a superclass to any number of subclasses**, just as you saw in Exercise 1 with the Vehicle class.

**Exercise 2**

Now create a **new package** called **exercise2**. **Copy** the four classes from Exercise 1 into this new package and **refactor** when prompted.

We now go back to a topic we haven’t thought about for a while, our old friend **input validation**! You see, for example, that none of the mutators you have written for Exercise 1 earlier have asked for any validation to be included, which naturally means that the state of Car or Bicycle objects could easily end up in an “**inconsistent state**”, which should really be avoided if possible.

In the case of the registration number for the Car class, you can literally enter any String value you like and it will be accepted, no questions asked. You are now going to try to rectify this situation by coding an input validation routine called **isValidRegistrationNumber**(). This method will take a String argument, representing the value entered by the user, and **return a boolean** true or false result.

This method will be called within the setRegistrationNumber() method of course, and if the method returns true i.e. the registration number was valid, then the attribute registrationNumber should get set to this value. However, if it is not valid then the registrationNumber attribute should be set to “**Invalid Registration Number**” instead (as seen in the code supplied below).

The next question is, where will be place the isValidRegistrationNumber() method. It could actually be placed into the Car class as a **private** method, that would act as a **utility method** for the setRegistrationNumber() mutator. However, this method is useful and we might like to use it in the future in other applications we could create. If we were to put it into the Car class as a **public** instance method, then we would need to create a Car object whenever we wished to use it, which would be quite awkward.

Therefore, I suggest that you create a brand new class called **Validator**, which is where your validation method will go. Create it as a **public static** method. This means that you can now call the method from anywhere you like – the **setRegistrationNumber**() method will now look as follows:

public void setRegistrationNumber(String reg) {  
 if(Validator.*isValidRegistrationNumber*(reg))  
 registrationNumber=reg;  
 else  
 registrationNumber="Invalid Registration Number";  
 }

The algorithm for determining whether a registration number is valid is the same one we used last year as follows:

* It must be between 6 and 12 characters long (inclusive)
* The first 2 characters of the registration plate must be digits that represent the year of purchase
* The third character must be a dash symbol,
* The next 2 characters must either be 2 uppercase letters or else a single uppercase letter followed by a dash, to represent the county in which the car was purchased. If the 4th and 5th characters were both uppercase letters, then the 6th character must be a dash, but if the 4th and 5th characters were an uppercase letter followed by a dash then the 6th character must be a digit
* Beyond the 6th character, all the remaining characters in the registration plate must consist of digits - as there could be any number of digits, up to a maximum of 6 digits, for this part of the registration plate, you will need a **loop** to process these characters

Examples of valid registration plates would be 08-KY-1234 and 97-L-985671, while examples of invalid ones would be 2008-C-12, 89-LKK-234 and 67\*WD\*6537

The difference compared with last year here is that, at the moment, the **user will only get one shot at entering a valid registration number**, if they don’t, the output for the car reg will just say “**Invalid Registration Number**”. This is because the mutators are only called **indirectly** in our driver when an object is being created i.e. when a constructor has been called and, because a **mutator is not meant to return a value**, we currently have no “easy” way of communicating a problem back to the driver program when an invalid reg has been supplied. We will deal with this later in the module though, within the important “**Exception Handling**” topic.

**Some Important OOP Terms Covered in this Lab Sheet**

You **need to be able to recognise and explain various OO features, concepts and terms** (for the mini-project presentation and, more importantly, for the final written exam). It is important to be able to explain various OO concepts well because you could easily be asked for such explanations at **interviews** in year 3 and for **job applications** generally going forward. In a nutshell, you need to be able to “speak” in OO terms.

Therefore, this section is designed to give some definitions/explanations for some of the concepts introduced in this lab sheet.

**Inheritance** is a form of **software reuse** that enables us to create new classes which are based on existing ones. The new class, called the **derived class** or **subclass**, will subsume the members of the existing class and can directly access all of the **non-private attributes and methods** (bar constructors) which are defined in the existing class, called the **base class** or **superclass**. The subclass then goes on to **add some of its own attributes and methods** to distinguish it from its superclass. It promotes **extensibility** by allowing us to “extend” the functionality of existing classes, **reduces code duplication** and allows **less complex classes** to be created.

**extends keyword** – the **extends** keyword in Java implies an **inheritance relationship** between two classes. The class name on the **left** of the keyword is the **subclass** and the one on the **right** is the **superclass**.

**super reference** – this is a **reference to the superclass object**. **super** is a Java keyword also. It can be used within a subclass to make an explicit call to the superclass constructor via super() or it can call a superclass method via **super.methodName**(). When a call to a superclass constructor is made via **super**(), it **can only be done from within a subclass constructor** and must be the very first line of code within that constructor.

**protected access modifier** – this access modifier can be applied to attributes and methods only and such **attributes and methods are only available to other classes within the same package** or from **subclasses** of the class in which they are defined. Their existence does **violate the principle of encapsulation** and so are rarely seen in practice.

**final methods and classes** – **final methods cannot be overridden** by a subclass while **final classes cannot be inherited** from. Some Java API classes are defined to be final e.g. String and Math.

**Single Inheritance** – Java only supports single inheritance meaning that any **class can only directly inherit from one immediate superclass**.